Embedded Systems with Linux

Working with the TT01 Cape

Manuel Domínguez-Pumar

Embedded Systems with Linux series

Volume 2: Working with the TT01 Cape

Manuel Domínguez Pumar

Electronic Engineering Department

Technical University of Catalonia – BarcelonaTech

First Edition – July 2018

ISBN 978-84-09-03814-5

AMG Editions, Barcelona, Spain

Artwork: JP Productions

Index

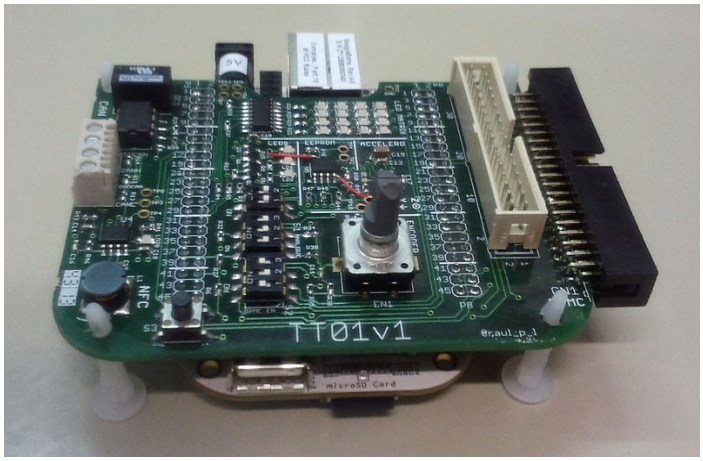
1 Introduction

The TT01 Cape board has been designed specifically for this laboratory. The purpose of this cape board is to provide a tool to learn how to program some of the most common peripherals used in embedded systems.

The peripherals implemented in the TT01 cape board are:

- General purpose LEDs.
- Push Button.
- Analog inputs.
- LED matrix controlled by a shift register.
- Rotary encoder.
- XYZ accelerometer.
- CAN bus.
- General Purpose Memory Controller (GPMC) connection.
- Near Field Communication (NFC) interface.
- Cape EEPROM.

All these peripheral resources are accessed from the Beaglebone through a set of signals available in the two 46-pin connection headers P8 and P9. Appendix I contains a detailed description of P8 and P9 header signals.



The aim of this laboratory module is to work with some of these peripherals. Although the next sections provide basic descriptions of the devices, you can find more information about the TT01 Cape Board hardware in the document "TT01_HWDescription_v1.0.pdf".

Prior to start the laboratory work, download the file *lm10.tar* from atenea and expand it in the directory *~/dsx/cape*. Six project directories, *lm10-GPIO1*, *lm10-GPIO2*, *lm10-LEDMX*, *lm10-ANALOG*, *lm10-ENCOD* and *lm10-ACCMTR* are created.

2 GPIO devices

2.1 General Purpose Input/Output

The Beaglebone has 128 General Purpose Inputs/Outputs (GPIOs). Some of them are allocated for specific functions, i.e. Beaglebone LEDs, but up to 66 GPIOs are accessible via the expansion headers P8 and P9.

Each pin of the Beaglebone processor (AM3359) has up to eight different function modes, which are selected with a multiplexer. You can find the modes available for each processor pin in the BeagleBone System Reference Manual, chapter 6.13 *Expansion Headers*.

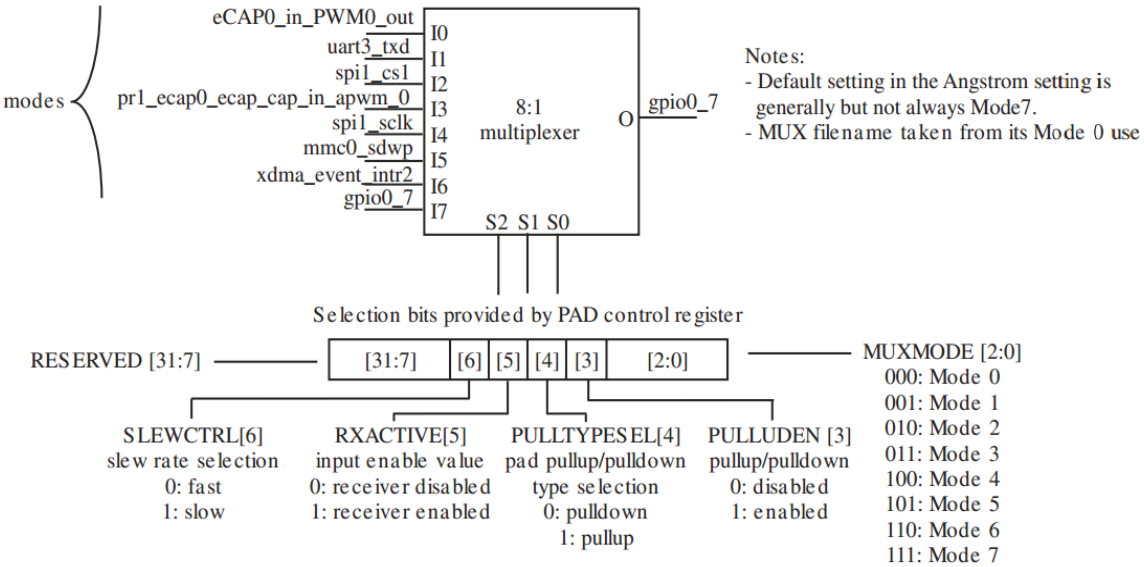
Each processor pin has associated a 32-bit PAD Control Register that defines the pin mode and attributes. Among others, the PAD control register includes the following fields:

Slew Rate Selection	SLEWCTRL	0 : fast, 1 : slow
Receiver input enabled	RXACTIVE	0 : disabled, 1 : enabled
Pad pull-up type selection	PULLTYPESEL	0 : pull-down, 1 : pull-up
Pull-up / pull-down selection	PULLEDEN	0 : disabled, 1 : enabled
Multiplexer mode selection	MUXMODE	000 : Mode 0, 001 : Mode 1, ... , 111 : Mode 7

As an example, the figure below describes the GPIO0_7 pin:

from Table 12, 13: Connector P9 Signal Pin Mux Options [Coley]

Pin	Proc	Signal Name	MODE0	MODE1	MODE2	MODE3	MODE4	MODE5	MODE6	MODE7
42	C18	GPIO0_7	eCAP0_in_PWM0_out	uart3_txd	spi1_cs1	pr1_ecap0_ecap_...	spi1_sclk	mmc0_sdwp	xdma_event_intr2	gpio0_7



A full description of the PAD control register can be found in chapter 9.3.1.51 of the Texas Instrument Technical Reference Manual for the AM335X Cortex-A8 Microprocessor.

On the other hand, Ubuntu Linux has a virtual file system, called `sysfs`, that provides individual access to the pins. In this way, each PAD control register can be accessed as an individual file.

In order to identify which is the file that corresponds to a specific GPIO pin, we must know its pin number. The general expression of a GPIO pin name is:

```
GPIO<bank number>_<pin number within bank>
```

GPIOs are grouped in four banks, each one with the following offset:

- bank 0, offset 0
- bank 1, offset 32
- bank 2, offset 64
- bank 3, offset 96

According to this, the pin number of a GPIO is calculated as:

```
pin number = (bank number * 32) + pin number within bank
```

As an example, the pin numbers of GPIO1_16 and GPIO2_10 are, respectively:

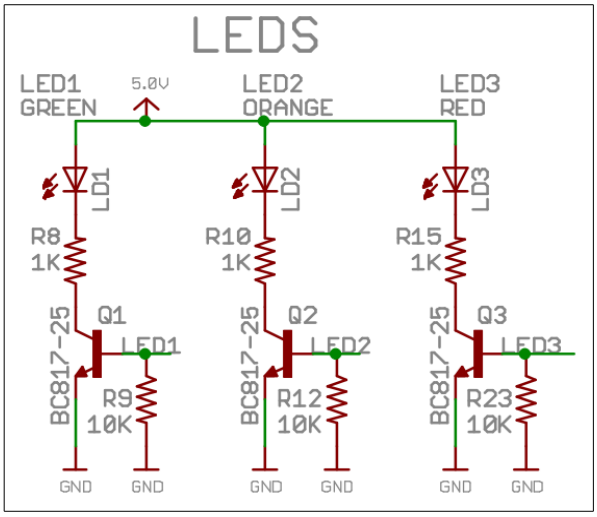
```
pin number = ( 1 * 32 ) + 16 = 48
pin number = ( 2 * 32 ) + 10 = 74
```

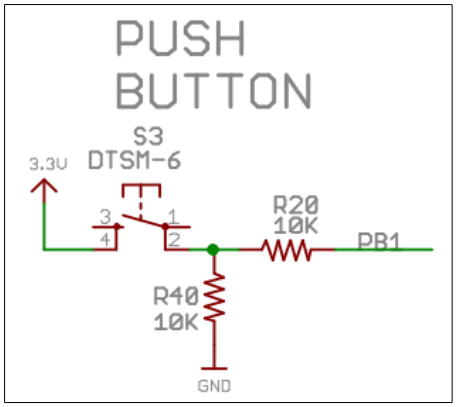
Summarizing, in order to properly configure GPIOs, we must identify which pins are used in the Beaglebone, which pins are available for the expansion boards, the modes of all those pins and their configuration registers. This information is found in the following documents:

- Reference Manual for the AM335X Cortex-A8 Microprocessor.
- Beaglebone schematic.
- Beaglebone reference manual.

2.2 GPIO LEDs and Pushbutton

The TT01 Cape has 3 LEDs and a pushbutton connected to GPIO pins.





These devices are controlled by the signals and located in the GPIO pins specified as follows:

Device	Control Signal	P9 BBone PIN	GPIO	GPIO Number
Green LED	LED1	15	GPIO1_16	48
Orange LED	LED2	16	GPIO1_19	51
Red LED	LED3	14	GPIO1_18	50
Pushbutton	PB1	41	GPIO0_20	20

2.3 Laboratory work

2.3.1 Control with Linux commands

As explained above, the processor pins are seen as files of a virtual file system. For instance, the kernel provides the directory `/sys/class/gpio` to control and access the GPIOs. This means that those pins can be accessed easily through Linux shell commands such as `echo` and `cat`.

As an example, let us open a Beaglebone terminal and create the files to configure GPIO1_16, which has pin number 48. First, the pin number must be sent (written) to the `export` file as follows:

```
beaglebone$ echo 48 > /sys/class/gpio/export
```

This creates the directory `gpio48`, which contains a set of files that are used to control the pin. In order to see this, execute the following commands:

```
beaglebone$ cd /sys/class/gpio
beaglebone$ ls -al
  total 0
  drwxr-xr-x  2 root root    0 Jan  1 00:00 .
  drwxr-xr-x 48 root root    0 Jan  1 00:00 ..
  --w-------  1 root root 4096 Jan  1 00:16 export
  lrwxrwxrwx  1 root root    0 Jan  1 00:16 gpio48 ->
```

```
                        ../../devices/virtual/gpio/gpio48
lrwxrwxrwx  1 root root    0 Jan  1 00:00 gpiochip0 ->
                        ../../devices/virtual/gpio/gpiochip0
lrwxrwxrwx  1 root root    0 Jan  1 00:00 gpiochip32 ->
                        ../../devices/virtual/gpio/gpiochip32
lrwxrwxrwx  1 root root    0 Jan  1 00:00 gpiochip64 ->
                        ../../devices/virtual/gpio/gpiochip64
lrwxrwxrwx  1 root root    0 Jan  1 00:00 gpiochip96 ->
                        ../../devices/virtual/gpio/gpiochip96
--w-------  1 root root 4096 Jan  1 00:16 unexport
```

Note that the directory `/sys/class/gpio/gpio48` has been created. If we now look into its contents:

```
beaglebone$ cd /sys/class/gpio/gpio48
beaglebone$ ls -al
  total 0
  drwxr-xr-x 3 root root    0 Jan  1 00:16 .
  drwxr-xr-x 7 root root    0 Jan  1 00:00 ..
  -rw-r--r-- 1 root root 4096 Jan  1 00:17 active_low
  -rw-r--r-- 1 root root 4096 Jan  1 00:17 direction
  -rw-r--r-- 1 root root 4096 Jan  1 00:17 edge
  drwxr-xr-x 2 root root    0 Jan  1 00:17 power
  lrwxrwxrwx 1 root root    0 Jan  1 00:17 subsystem ->
                                  ../../../../class/gpio
  -rw-r--r-- 1 root root 4096 Jan  1 00:16 uevent
  -rw-r--r-- 1 root root 4096 Jan  1 00:17 value
```

The files `direction` and `value` are used for basic configuration of the GPIO pin.

The configuration options available for each GPIO pin are described in the *Linux GPIO Interfaces Manual*, available at `https://www.kernel.org/doc/Documentation/gpio/`

For instance, to configure GPIO1_16 (pin 48) as an output pin with logic value 1, we should execute:

```
beaglebone$ echo out > /sys/class/gpio/gpio48/direction
beaglebone$ echo 1 > /sys/class/gpio/gpio48/value
```

Let us now write a shell script that switches the LED LD2 every second. This LED is connected to the pin 48 (GPIO1_16). The contains of the script are the following:

```
#!/bin/bash
if [ ! -d /sys/class/gpio/gpio48 ]; then echo 48 > /sys/class/gpio/export; fi
echo out > /sys/class/gpio/gpio48/direction
while [ 1 ]; do
    echo 1 > /sys/class/gpio/gpio48/value
    sleep 1
    echo 0 > /sys/class/gpio/gpio48/value
    sleep 1
done
```

Go to the Beaglebone home root directory (`cd /home/root`) and save there the script file, with name `LD2.sh`.

Once done, change the script rights and execute it:

```
beaglebone$ chmod u=+rwx LD2.sh
beaglebone$ ./LD2.sh
```

2.3.2 Control with C

Let us now perform the GPIO control using C code and file system calls: open, read, write, close... The following example (GPIO1.c) switches ON the LED connected to GPIO1_16 for 3 seconds:

```
//****************************************************************************
// GPIO1.c - Switch ON LD1- GPIO1_16 - GPIO48 for 3 seconds
//****************************************************************************

#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define output "out"
#define input "in"
#define pin_number "48"
#define logic_high "1"
#define export_file         "/sys/class/gpio/export"
#define direction_file      "/sys/class/gpio/gpio48/direction"
#define value_file          "/sys/class/gpio/gpio48/value"
#define unexport_file       "/sys/class/gpio/unexport"

int main(void)
{
      // file descriptors
      int gpio48_export;
      int gpio48_unexport;
      int gpio48_value;
      int gpio48_direction;

      // write pin variable
      int write_pin;

      // open the export file
      gpio48_export = open(export_file, O_WRONLY);
      if (gpio48_export == -1) {
            perror("Unable to open gpio48 export\n");
      }
```

```
        // enable the gpio48 pin
        write_pin = write(gpio48_export, pin_number, strlen(pin_number));
        if (write_pin == -1) {
                perror("Unable to write gpio48 export\n");
        }

        // open gpio48 direction file
        gpio48_direction = open(direction_file, O_WRONLY);
        if (gpio48_direction == -1) {
                perror("Unable to open gpio48 direction\n");
        }

        // configure gpio48 as output
        write_pin = write(gpio48_direction, output, strlen(output));
        if (write_pin == -1) {
                perror("Unable to configure gpio48 as output\n");
        }

        // open gpio48 value file
        gpio48_value = open(value_file, O_WRONLY);
        if (gpio48_value == -1) {
                perror ("Unable to open gpio48 value\n");
        }

        // set logic 1 to gpio48 = switch ON the LED
        write_pin = write (gpio48_value, logic_high, strlen(logic_high));
        if (write_pin == -1) {
                perror("Unable to set gpio48 to logic high\n");
        }

        // sleep for 3 seconds
        sleep(3);

        // open unexport file
        gpio48_unexport = open(unexport_file, O_WRONLY);
        if (gpio48_unexport == -1) {
                perror("Unable to open unexport\n");
        }

        // write pin number in unexport file = disable gpio48 pin
        write_pin = write(gpio48_unexport, pin_number, strlen(pin_number));
        if (write_pin == -1) {
                perror("Unable to write gpio48 unexport \n");
        }

        // close all files and exit
        close(gpio48_export);
        close(gpio48_unexport);
        close(gpio48_direction);
        close(gpio48_value);
        return 1;
}
```

The next example (GPIO2.c) toggles the three LEDs every second. The program exits when the push button is pressed for 1 second or more.

```
//****************************************************************************
// GPIO2.c - Toggle LEDs every second. Break with push button ON > 1 sec
//****************************************************************************

#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define output "out"
#define input "in"

#define pin_48 "48" // LED1 - LD1 - GPIO1_16
#define pin_51 "51" // LED2 - LD2 - GPIO1_19
#define pin_50 "50" // LED3 - LD3 - GPIO1_18
#define pin_20 "20" // PUSH BUTTON - PB1 – GPIO0_20

#define export_file       "/sys/class/gpio/export"
#define unexport_file   "/sys/class/gpio/unexport"
#define gpio48_dir_file "/sys/class/gpio/gpio48/direction"
#define gpio51_dir_file "/sys/class/gpio/gpio51/direction"
#define gpio50_dir_file "/sys/class/gpio/gpio50/direction"
#define gpio20_dir_file "/sys/class/gpio/gpio20/direction"
#define gpio48_val_file "/sys/class/gpio/gpio48/value"
#define gpio51_val_file "/sys/class/gpio/gpio51/value"
#define gpio50_val_file "/sys/class/gpio/gpio50/value"
#define gpio20_val_file "/sys/class/gpio/gpio20/value"

int main(void)
{
      // file descriptors
      int gpio_exp;
      int gpio_unexp;
      int gpio48_value;
      int gpio48_direction;
      int gpio51_value;
      int gpio51_direction;
      int gpio50_value;
      int gpio50_direction;
      int gpio20_value;
      int gpio20_direction;

      int read_size;   // read buffer size
      int write_size;  // write buffer size

      // variable to toggle LEDs value
      char LED[] = "0";

      // variable for pushbutton status: 0 - not pushed, 1 - pushed
      char value_pb[] = "0";

      // open gpio export file
      gpio_exp = open(export_file, O_WRONLY);
```

```
if (gpio_exp == -1) {
        perror("Unable to open gpio export\n");
}

// enable gpio48 pin
write_size = write(gpio_exp, pin_48, strlen(pin_48));
if (write_size == -1) {
        perror("Unable to write gpio48 export\n");
}
// open gpio48 direction file
gpio48_direction = open(gpio48_dir_file, O_WRONLY);
if (gpio48_direction == -1) {
        perror("Unable to open gpio48 direction\n");
}
// configure gpio48 as output
write_size = write(gpio48_direction, output, strlen(output));
if (write_size == -1) {
        perror("Unable to configure gpio48 as output\n");
}
// open gpio48 value file
gpio48_value = open(gpio48_val_file, O_WRONLY);
if (gpio48_value == -1) {
        perror ("Unable to open gpio48 value\n");
}

// enable gpio51 pin
write_size = write(gpio_exp, pin_51, strlen(pin_51));
if (write_size == -1) {
        perror("Unable to write gpio51 export\n");
}
// open gpio51 direction file
gpio51_direction = open(gpio51_dir_file, O_WRONLY);
if (gpio51_direction == -1) {
        perror("Unable to open gpio51 direction 51\n");
}
// configure gpio51 as output
write_size = write(gpio51_direction, output, strlen(output));
if (write_size == -1) {
        perror("Unable to configure gpio51 as output\n");
}
// open gpio51 value file
gpio51_value = open(gpio51_val_file, O_WRONLY);
if (gpio51_value == -1) {
        perror ("Unable to open gpio51 value\n");
}

// enable gpio50 pin
write_size = write(gpio_exp, pin_50, strlen(pin_50));
if (write_size == -1) {
        perror("Unable to write gpio50 export\n");
}
// open gpio50 direction file
gpio50_direction = open(gpio50_dir_file, O_WRONLY);
if (gpio50_direction == -1) {
        perror("Unable to open gpio50 direction\n");
}
// configure gpio50 as output
write_size = write(gpio50_direction, output, strlen(output));
```

```
        if (write_size == -1) {
                perror("Unable to configure gpio50 as output\n");
        }
        // open gpio50 value file
        gpio50_value = open(gpio50_val_file, O_WRONLY);
        if (gpio50_value == -1) {
                perror ("Unable to open gpio50 value\n");
        }

        // enable gpio20 pin
        write_size = write(gpio_exp, pin_20, strlen(pin_20));
        if (write_size == -1) {
                perror("Unable to write gpio20 export\n");
        }
        // open gpio20 direction file
        gpio20_direction = open(gpio20_dir_file, O_WRONLY);
        if (gpio20_direction == -1) {
                perror("Unable to open gpio20 direction\n");
        }
        // configure gpio20 as input
        write_size = write(gpio20_direction, input, strlen(input));
        if (write_size == -1) {
                perror("Unable to configure gpio20 as input\n");
        }

        // open gpio20 value file
        gpio20_value = open(gpio20_val_file, O_RDONLY);
        if (gpio20_value == -1) {
                perror ("Unable to open gpio20 value\n");
        }

        // Loop to toggle LEDs every second
        while(1) {
                // change the value of LED
                if (strcmp(LED,"1") == 0)
                        strcpy(LED,"0");
                else
                        strcpy(LED,"1");

                // set gpio48 value = LED
                write_size = write(gpio48_value, LED, strlen(LED));
                if (write_size == -1) {
                        perror("Unable to set new gpio48 value\n");
                }
                // set gpio51 value = LED
                write_size = write(gpio51_value, LED, strlen(LED));
                if (write_size == -1) {
                        perror("Unable to set new gpio51 value\n");
                }
                // set gpio50 value = LED
                write_size = write(gpio50_value, LED, strlen(LED));
                if (write_size == -1) {
                        perror("Unable to set new gpio50 value\n");
                }

                // read gpio20 (push button) value
                lseek(gpio20_val_file, 0, SEEK_SET);
                read_size = read(gpio20_value, value_pb, 1);
```

```
                if (read_size == -1) {
                        perror("Unable to read push button value\n");
                }

                printf("logic status pushbutton %s\n", value_pb);

                // if pushbutton pressed > 1 sec then break the while loop
                if (strcmp(value_pb,"1") == 0)
                        break;

                sleep(1);  // wait for 1 second
        }

        // open unexport file
        gpio_unexp = open(unexport_file, O_WRONLY);
        if (gpio_unexp == -1) {
                perror("Unable to open gpio48 unexport\n");
        }
        // disable gpio48 pin
        write_size = write(gpio_unexp, pin_48, strlen(pin_48));
        if (write_size == -1) {
                perror("Unable to write gpio48 unexport\n");
        }

        // disable gpio51 pin
        write_size = write(gpio_unexp, pin_51, strlen(pin_51));
        if (write_size == -1) {
                perror("Unable to write gpio51 unexport\n");
        }

        // disable gpio50 pin
        write_size = write(gpio_unexp, pin_50, strlen(pin_50));
        if (write_size == -1) {
                perror("Unable to write gpio50 unexport\n");
        }

        // disable gpio20 pin
        write_size = write(gpio_unexp, pin_20, strlen(pin_20));
        if (write_size == -1) {
                perror("Unable to write gpio20 unexport\n");
        }

        // close all files and exit
        close(gpio_exp);
        close(gpio_unexp);
        close(gpio48_direction);
        close(gpio48_value);
        close(gpio51_direction);
        close(gpio51_value);
        close(gpio50_direction);
        close(gpio50_value);
        close(gpio20_direction);
        close(gpio20_value);

        return 0;
}
```

2.3.2.1 Proposed exercises

Write a C program implementing an up counter for integer numbers from 0 to 7. The counter value must be displayed in binary code on the LEDs according to the table below, being BIT2 the MSB and BIT0 the LSB. The counter increases its value every second. The program exits only when the push button is pressed being all LEDs OFF (0 value).

BIT 2	BIT 1	BIT 0
LD3	LD1	LD2

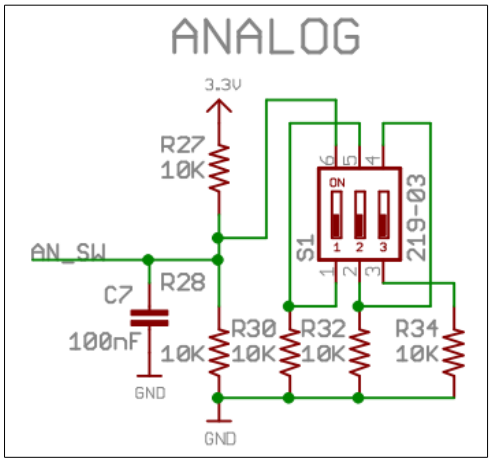
3 Analog input

3.1 TT01 Cape ADC

The Beaglebone has a 12-bit successive-approximation ADC with 8 channels. The reference voltage is 1.8V, so that the voltage equivalent to the ADC counts lecture is:

```
voltage = (number_counts_ADC / (2^12)) * 1.8
```

In the TT01 Cape, the analog signal AN_SW is connected to the signal AIN0 (the ADC 0 input channel) of the Beaglebone. The signal AN_SW is generated by the resistive voltage divider shown below, then its value is configurable from the three switches S1 as shown in the table.



1	2	3	AN_SW [V]
OFF	-	-	1.6
ON	OFF	-	1.1
ON	ON	OFF	0.8
ON	ON	ON	0.7

3.2 Laboratory work

3.2.1 Control with C

As in the GPIOs case, the ADC channels are available through the file system, now being

```
/sys/bus/platform/devices/tiadc/iio:device0/in_voltage0_raw
```

the file where the count value provided by the ADC channel 0 can be read. This value ranges between 0 and 4095, so it is read as a 4 character string.

The program ANALOG.c listed below shows the value of the ADC channel 0, in Volts, every 2 seconds. To exit the program, the push button must be pressed for at least 2 seconds.

```
//****************************************************************************
// ANALOG.c - Read ADC0 every 2 seconds. Pushbutton > 1 sec to exit
//****************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <math.h>

// analog file ADC0 path
#define ain0_in "/sys/bus/platform/devices/tiadc/iio:device0/in_voltage0_raw"

#define input "in"
#define pin_20 "20"   // PUSH BUTTON - PB1 - GPIO0_20
#define export_file         "/sys/class/gpio/export"
#define unexport_file       "/sys/class/gpio/unexport"
#define gpio20_dir_file     "/sys/class/gpio/gpio20/direction"
#define gpio20_val_file     "/sys/class/gpio/gpio20/value"

int main (void)
{
      // file descriptors
      int an0_file;
      int gpio_exp;
      int gpio_unexp;
      int gpio20_value;
      int gpio20_direction;
      int read_size;    // read buffer size
      int write_size;   // write buffer size

      char value_pb[] = "0"; // pushbutton status: 0 - not pushed; 1 - pushed

      // variables for the ADC0 value
      char an0_val[4];        // as read (string)
      int counts_value;       // as integer (0 to 4095)
      float volts_value;      // as voltage (Volts)

      // open gpio export file
      gpio_exp = open(export_file, O_WRONLY);
      if (gpio_exp == -1) {
            perror("Unable to open gpio export\n");
      }
      // enable gpio20 pin
      write_size = write(gpio_exp, pin_20, strlen(pin_20));
      if (write_size == -1) {
            perror("Unable to write gpio export\n");
      }
      // open gpio20 direction file
      gpio20_direction = open(gpio20_dir_file, O_WRONLY);
      if (gpio20_direction == -1) {
            perror("Unable to open gpio20 direction\n");
      }
```

```
// configure gpio20 as input
write_size = write(gpio20_direction, input, strlen(input));
if (write_size == -1) {
        perror("Unable to configure gpio20 as input\n");
}

// Open ADC0 analog input file
if ((an0_file = open(ain0_in, O_RDONLY)) < 0) {
        perror("Unable to open analog input\n");
        exit(1);
}

// open gpio20 value file
gpio20_value = open(gpio20_val_file, O_RDONLY);
if (gpio20_value == -1) {
        perror ("Unable to switch on the GPIO20\n");
}

// loop to read ADC0 value every 2 seconds
while(1) {

        // Read ADC0 value as string
        lseek(an0_file, 0, SEEK_SET);
        if ((read_size = read(an0_file, an0_val, sizeof(an0_val))) < 0) {
                perror("Unable to read analog value\n");
                exit(1);
        }

        // Add NULL string terminator and display read value
        an0_val[read_size]='\0';
        printf("AINO reading [0 to 4095]: %s\n", an0_val);

        // Convert ADC0 value to integer
        counts_value = atoi(an0_val);

        // Convert ADC0 value to volts and display it
        volts_value = ((counts_value/pow(2,12)) * 1.8);
        printf("AINO voltage [V]: %.*f\n\n", 2, volts_value);

        // read gpio20 push button
        lseek(gpio20_value, 0, SEEK_SET);
        read_size = read(gpio20_value, value_pb,1);
        if (read_size == -1) {
                perror("Unable to read input value\n");
        }
        printf("keep pushbutton pressed to exit\n\n");
        // if pushbutton pressed > 1 sec, break the while loop
        if( strcmp(value_pb,"1") == 0)
                break;

        sleep(2);  // Wait for 2 seconds
}

// open gpio unexport file
gpio_unexp = open(unexport_file, O_WRONLY);
if (gpio_unexp == -1) {
        perror("Unable to open unexport gpio20\n");
}
```

```
        // disable gpio20 pin
        write_size = write(gpio_unexp, pin_20, strlen(pin_20));
        if (write_size == -1) {
                perror("Unable to write gpio unexport\n");
        }

        // close all files and exit
        close(gpio_exp);
        close(gpio_unexp);
        close(gpio20_direction);
        close(gpio20_value);
        close(an0_file);
        return 0;
}
```

3.2.2 Proposed exercises

Write a C program to display the voltage range of the analog input on the TT01 LEDs, according to the following table:

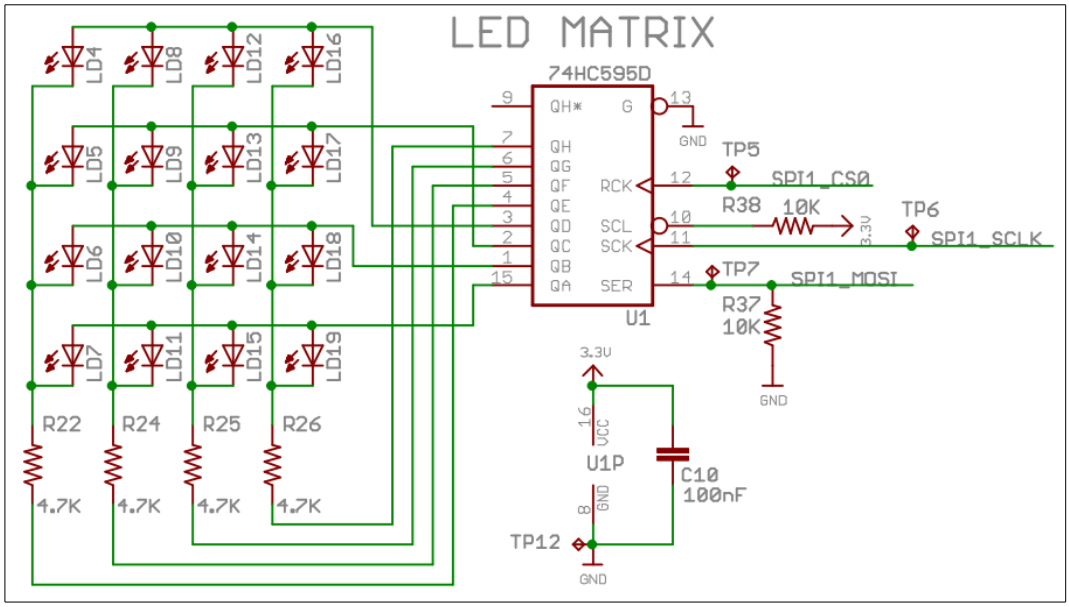
ANALOG INPUT	LD3	LD1	LD2
1.8 > V > 1.65	0	0	1
1.65 > V > 1.1	0	1	0
1.1 > V > 0.825	0	1	1
0.825 > V > 0.66	1	0	0
0.66 > V	1	0	1

4 LED Matrix

In user interfaces, connecting LEDs in a matrix topology, wired together in rows and columns, is a common practice to minimize the number of pins needed to drive them. This multiplexed mode drive requires some extra processing, but it is more efficient than driving directly each LED.

4.1 TT01 Cape LED Matrix

The LED matrix included in the TT01 Cape is controlled from a shift register (74HC595D). This shift register is accessed through the SPI1 (Serial Peripheral Interface) bus of the Beaglebone. See the schematic below.



A SPI is a bidirectional bus with the following signals:

- MOSI: Master Output Slave Input.
- MISO: Master Input Slave Output.
- SCLK: Serial Clock
- SS: Slave Select

The correspondence of the SPI bus signals to the ones connected to the shift register in the TT01 Cape is the following:

SPI Signal	TT01 Signal
MOSI	SPI1_MOSI
MISO	Not connected
SCLK	SPI1_SCLK
SS	SPI1_CS0

Note that, since the shift register works only as slave/receiver, the MISO signal becomes useless and so it is left unconnected.

4.2 Laboratory work

4.2.1 Control with C

Unlike the GPIOs, the SPI bus interfaces are high level resources that are available through driver files located in the /dev directory.

The program LEDMX.c toggles all LEDs of the LED matrix every 2 seconds. To exit the program, keep the push button pressed for at least 2 seconds. Note also that, in order to improve the code, a set of functions to handle GPIO files is used in this example.

```
//****************************************************************************
// LEDMX.c - Switch on/off all Matrix LEDs every 2 seconds.
//                To exit, press push button > 2 seconds
//****************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

// function prototypes
int gpio_export(char *gpio_num);
int gpio_set_direction (char *gpio_num, char *dir );
int gpio_read_value (char *gpio_num, char *value);
int gpio_unexport (char *gpio_num);

#define input "in"
#define pin_20 "20" // PUSH BUTTON - PB1 - GPIO0_20
#define sys_gpio_dir  "/sys/class/gpio/"
#define spi_file_out   "/dev/spidev1.0" // SPI file path

int main(void)
{
      int spi_file; // SPI file descriptor
      int spi_wr;  // SPI write integer return

      char wr_buf[2] = {0x0F,0x00};  // register values to toggle all Matrix LEDs
      char pb_state[] = "0";              // push button status
      int matrix = 0;                     // LED matrix status

      // export GPIO20 push button and configure as input
      gpio_export(pin_20);
      gpio_set_direction(pin_20, input);

      // open SPI file
      if ((spi_file = open(spi_file_out, O_RDWR)) < 0) {
            perror("Unable to open SPI file\n");
            exit(1);
      }

      // loop to toggle the LED Matrix
      while(1) {
            // write in the SPI file
            if ((spi_wr = write(spi_file, &wr_buf[matrix], sizeof(char))) < 0) {
                  perror("Unable to write in SPI file\n");
                  exit(1);
            }

            // wait 2 seconds to change LED matrix status
            sleep(2);

            // change buffer index to change LED Matrix status
            if (matrix == 0) {
                  matrix = 1;
                  printf ("Switch ON LEDMATRIX\n\n");
            }
            else {
```

```
                        printf ("Switch OFF LEDMATRIX\n\n");
                        matrix = 0;
                }

                // check push button status
                gpio_read_value (pin_20, pb_state);
                // if push button is pressed, then finish the program
                if ( strcmp(pb_state,"1") == 0) {
                        printf ("Push button pressed\nExit\n");
                        break;
                }
        }
        // close SPI file
        close(spi_file);

        // unexport GPIO20 push button
        gpio_unexport(pin_20);

        return 0;
}

//  ***************************************************************************
//    Function gpio_export
//       gpio_num: GPIO pin number
//  ***************************************************************************
int gpio_export(char *gpio_num) {

        int exp_fd, write_size;

        // open GPIO export file
        exp_fd = open(sys_gpio_dir"/export", O_WRONLY);
        if (exp_fd == -1) {
                perror("Unable to open export\n");
        }
        // enable GPIO pin
        write_size = write(exp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write export\n");
        }
        close (exp_fd);
        return 0;
}

//  ***************************************************************************
//    Function gpio_set_direction
//       gpio_num:   GPIO pin number
//       dir:            GPIO direction "in"/"out"
//  ***************************************************************************
int gpio_set_direction (char *gpio_num, char *dir) {

        int dir_fd, write_size;
        char buffer[64];
        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/direction", gpio_num);

        // open GPIO direction file
        dir_fd = open(buffer, O_WRONLY);
        if (dir_fd == -1) {
                perror("Unable to open direction file\n");
```

```
        }
        // configure GPIO direction
        write_size = write(dir_fd, dir, strlen(dir));
        if (write_size == -1) {
                perror("Unable to configure the direction\n");
        }
        close (dir_fd);
        return 0;
}

//  ****************************************************************************
//    Function: gpio_read_value
//       gpio_num:   GPIO pin number
//        value:        gpio value read
//  ****************************************************************************
int gpio_read_value (char *gpio_num, char *value) {

        int value_fd, read_size;
        char buffer[64];
        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/value", gpio_num);

        // open GPIO value file
        value_fd = open(buffer, O_RDONLY);
        if (value_fd == -1) {
                perror("Unable to open value file\n");
        }
        // read GPIO value file
        read_size = read(value_fd, value,1);
        if (read_size == -1) {
                perror("Unable to read input value\n");
        }
        close(value_fd);
        return 0;
}

//  ****************************************************************************
//    Function: gpio_unexport
//         gpio_num:  GPIO pin number
//  ****************************************************************************
int gpio_unexport (char *gpio_num) {

        int unexp_fd , write_size;

        // open GPIO unexport file
        unexp_fd = open(sys_gpio_dir"/unexport", O_WRONLY);
        if (unexp_fd == -1) {
                perror("Unable to open unexport\n");
        }
        // disable GPIO pin
        write_size = write(unexp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write unexport\n");
        }
        close(unexp_fd);
        return 0;
}
```

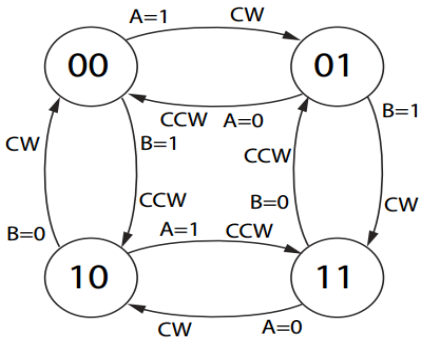
4.2.2 Proposed exercises

- Every second switch-on only one LED in the matrix, in an incremental way, from LD4 to LD19.
- Use the LED matrix to display the Analog volatge values provided by the S1 switches, as follows:

ANALOG INPUT	LEDS ON
1.8 > V > 1.65	All LEDS
1.65 > V > 1.1	LD4 LD5 LD6 LD8 LD9 LD10 LD12 LD13 LD14 LD16 LD17 LD18
1.1 > V > 0.825	LD4 LD5 LD8 LD9 LD12 LD13 LD16 LD17
0.825 > V > 0.66	LD4 LD8 LD12 LD16
0.66 > V	NONE

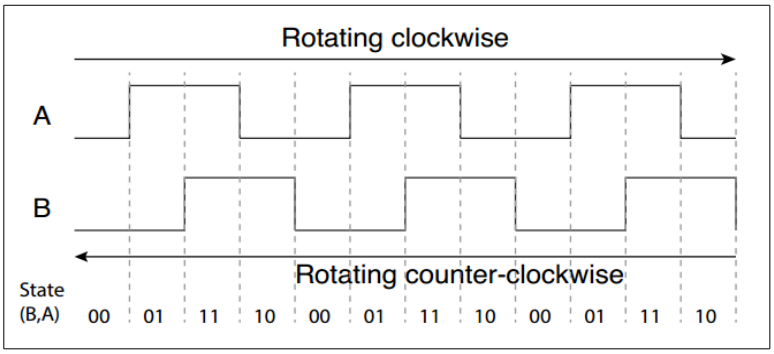
5 Rotary encoder

A rotary encoder is a peripheral typically used in user interfaces to navigate through the menus in displays. It has has 2 outputs (namely, A and B) 90º out of phase.



By monitoring the two outputs of a rotary encoder it is possible to identify the direction of the rotation, and therefore the relative position from an initial state.

The state machine shown on the left allows to know if the direction of the rotation is either clockwise (CW) or counter clockwise (CCW).



5.1 TT01 Cape Encoder

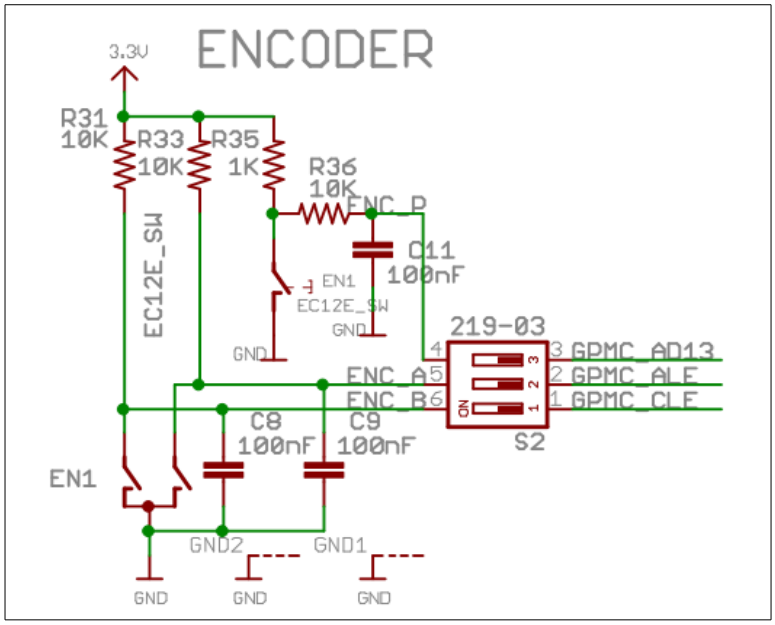
The TT01 Cape includes a rotary encoder with the output signals connected to GPIO pins as follows:

ENCODER NET	GPMC NET	BBONE GPIO	GPIO PIN
ENC_A	GPMC_ALE	GPIO2_2	66
ENC_B	GPMC_CLE	GPIO2_5	69

The rotary encoder has a third output, ENC_P, that indicates if the push button is being pressed. This feature is available only in some encoders.

The schematic below shows the implementation of the rotary encoder in the TT01 Cape. The bank of switches S2 selects between using the encoder outputs (ENC_A, ENC_B, ENC_P) or using the signals GPMC_AD13, GPMC_ALE and GPMC_CLE (see next Lab modules for details).

When GPMC signals must be used, set all S2 switches to OFF state. It disables the encoder monitoring.



5.2 Laboratory work

5.2.1 Control with C

There are different ways to read a rotary encoder, we are going to monitor it with interrupt inputs by both (up and down) edges. To do so, we will configure the GPIOs with those features and use the *poll()* system call, provided by Linux to implement Multiplexed Input / Output (I/O).

To understand the *poll* function, consider that until now we have used *read()* and *write()* system calls to work with files. These file I/O system calls may block (thus the process may remain stopped) until data is transferred. However, this may be not desirable in some cases, for instance:

- Checking whether an I/O function is possible or not on a file descriptor without blocking it.
- Monitoring multiple file descriptors to see if an I/O function is possible on any of them.

poll()

Purpose: notify, without blocking, when a file becomes ready to read or write.

```
#include <poll.h>

int poll (struct pollfd *fds, nfds_t nfds, int timeout);
```

The *nfds* parameter is the number of items in the *fds* structure.

The *timeout* parameter specifies the maximum time to wait in milliseconds, before returning regardless of any ready I/O. A negative value denotes infinite timeout. A value 0 instructs the call to return immediately, listing any file descriptors with pending ready I/O, but not to wait for any further events; in this way *poll()* is true to its name, polling once and immediately returning.

The structure *pollfd* is defined as follows:

```
#include <poll.h>

struct pollfd {
    int fd;         /* file descriptor */
    short events;   /* requested events to watch */
    short revents;  /* returned events witnessed */
};
```

- The field *fd* contains a file descriptor for an open file.
- The field *events* is an input: a bit mask specifying the events the application is interested in.
- The field *revents* is an output filled by the kernel with the events that actually occurred.

The bits that may be set in *events* and returned in *revents* are defined in *poll.h*:

- POLLIN: There is data to read.
- POLLPRI: There is urgent data to read.
- POLLOUT: Writing now will not block.
- POLLRDHUP: Stream socket peer closed connection, or shut down writing half of connection. The GNU SOURCE feature test macro must be defined in order to obtain this definition.
- POLLERR: Error condition (output only).
- POLLHUP: Hang up (output only).
- POLLNVAL: Invalid request: fd not open (output only).

The Multiplexed I/O is explained as follows in the book “Linux system programming” (O'Reilly):

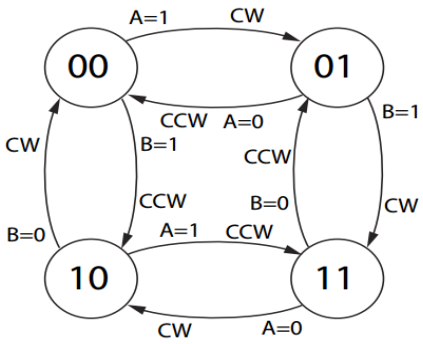
Multiplexed I/O allows an application to concurrently block on multiple file descriptors and receive notification when any one of them becomes ready to read or write without blocking. Multiplexed I/O thus becomes the pivot point for the application, designed similarly to the following activity:

1. *Multiplexed I/O: Tell me when a file descriptor becomes ready for I/O.*
2. *Nothing ready? Sleep until one or more file descriptors are ready.*
3. *Woken up! What is ready?*
4. *Handle all file descriptors ready for I/O, without blocking.*
5. *Go back to step 1.*

The ENCODER.c program displays the ENC_A and ENC_B status while you turn the encoder. The encoder pins ENC_A and ENC_B are configured as inputs with detection in both edges.

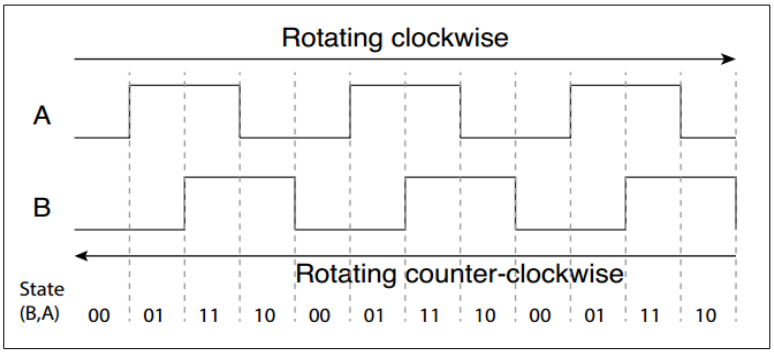
```
//***************************************************************************
// ENCODER.c - Displays the value of the encoder outputs
//***************************************************************************

#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <poll.h>Rotary encoder
```

A rotary encoder is a peripheral typically used in user interfaces to navigate through the menus in displays. It has has 2 outputs (namely, A and B) 90º out of phase.



By monitoring the two outputs of a rotary encoder it is possible to identify the direction of the rotation, and therefore the relative position from an initial state.

The state machine shown on the left allows to know if the direction of the rotation is either clockwise (CW) or counter clockwise (CCW).



5.3 TT01 Cape Encoder

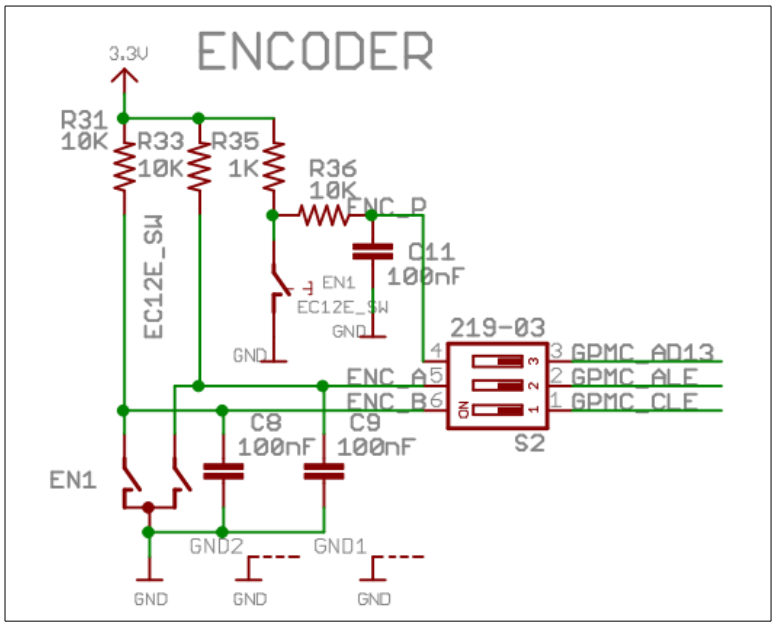
The TT01 Cape includes a rotary encoder with the output signals connected to GPIO pins as follows:

ENCODER NET	GPMC NET	BBONE GPIO	GPIO PIN
ENC_A	GPMC_ALE	GPIO2_2	66
ENC_B	GPMC_CLE	GPIO2_5	69

The rotary encoder has a third output, ENC_P, that indicates if the push button is being pressed. This feature is available only in some encoders.

The schematic below shows the implementation of the rotary encoder in the TT01 Cape. The bank of switches S2 selects between using the encoder outputs (ENC_A, ENC_B, ENC_P) or using the signals GPMC_AD13, GPMC_ALE and GPMC_CLE (see next Lab modules for details).

When GPMC signals must be used, set all S2 switches to OFF state. It disables the encoder monitoring.



5.4 Laboratory work

5.4.1 Control with C

There are different ways to read a rotary encoder, we are going to monitor it with interrupt inputs by both (up and down) edges. To do so, we will configure the GPIOs with those features and use the *poll()* system call, provided by Linux to implement Multiplexed Input / Output (I/O).

To understand the *poll* function, consider that until now we have used *read()* and *write()* system calls to work with files. These file I/O system calls may block (thus the process may remain stopped) until data is transferred. However, this may be not desirable in some cases, for instance:

- Checking whether an I/O function is possible or not on a file descriptor without blocking it.
- Monitoring multiple file descriptors to see if an I/O function is possible on any of them.

poll()

Purpose: notify, without blocking, when a file becomes ready to read or write.

```
#include <poll.h>

int poll (struct pollfd *fds, nfds_t nfds, int timeout);
```

The *nfds* parameter is the number of items in the *fds* structure.

The *timeout* parameter specifies the maximum time to wait in milliseconds, before returning regardless of any ready I/O. A negative value denotes infinite timeout. A value 0 instructs the call to return immediately, listing any file descriptors with pending ready I/O, but not to wait for any further events; in this way *poll()* is true to its name, polling once and immediately returning.

The structure *pollfd* is defined as follows:

```
#include <poll.h>

struct pollfd {
    int fd;         /* file descriptor */
    short events;   /* requested events to watch */
    short revents;  /* returned events witnessed */
};
```

- The field *fd* contains a file descriptor for an open file.
- The field *events* is an input: a bit mask specifying the events the application is interested in.
- The field *revents* is an output filled by the kernel with the events that actually occurred.

The bits that may be set in *events* and returned in *revents* are defined in *poll.h*:

- POLLIN: There is data to read.
- POLLPRI: There is urgent data to read.
- POLLOUT: Writing now will not block.
- POLLRDHUP: Stream socket peer closed connection, or shut down writing half of connection. The GNU SOURCE feature test macro must be defined in order to obtain this definition.
- POLLERR: Error condition (output only).
- POLLHUP: Hang up (output only).
- POLLNVAL: Invalid request: fd not open (output only).

The Multiplexed I/O is explained as follows in the book “Linux system programming” (O'Reilly):

Multiplexed I/O allows an application to concurrently block on multiple file descriptors and receive notification when any one of them becomes ready to read

or write without blocking. Multiplexed I/O thus becomes the pivot point for the application, designed similarly to the following activity:

1. *Multiplexed I/O: Tell me when a file descriptor becomes ready for I/O.*
2. *Nothing ready? Sleep until one or more file descriptors are ready.*
3. *Woken up! What is ready?*
4. *Handle all file descriptors ready for I/O, without blocking.*
5. *Go back to step 1.*
6.

The ENCODER.c program displays the ENC_A and ENC_B status while you turn the encoder. The encoder pins ENC_A and ENC_B are configured as inputs with detection in both edges.

```
//****************************************************************************
// ENCODER.c - Displays the value of the encoder outputs
//****************************************************************************

#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <poll.h>
#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <string.h>

// function prototypes
int get_value_lead(int value_fd);
int gpio_export(char *gpio_num);
int gpio_set_direction(char *gpio_num, char *dir );
int gpio_unexport(char *gpio_num);
int gpio_set_edge(char *gpio_num, char *edge);
int gpio_open_fd(char *gpio_num);

// program constants
#define A 0
#define B 1

#define pin_66 "66" // ENC_A
#define pin_69 "69" // ENC_B

#define input "in"
#define edge_both "both"

#define sys_gpio_dir "/sys/class/gpio"
#define MAX_BUF 64

int main(void)
{
   int gpio_val[2];
   struct pollfd poll_gpio_fd[2];
   int lead[2], ready;
```

```
    // export GPIO66 (ENCA) and GPIO69 (ENCB)
    gpio_export(pin_66);
    gpio_export(pin_69);

    // set input direction to GPIO66 and GPIO69
    gpio_set_direction (pin_66, input );
    gpio_set_direction (pin_69, input );

    // set edge of GPIO66 and GPIO69 to both
    gpio_set_edge (pin_66, edge_both);
    gpio_set_edge (pin_69, edge_both);

    // open GPIO66 value file
    gpio_val[A] = gpio_open_fd(pin_66);
    if (gpio_val[A] < 0) {
            perror("Unable to open GPIO66 value file\n");
            exit (1);
    }

    // open GPIO69 value file
    gpio_val[B] = gpio_open_fd(pin_69);
    if (gpio_val[B] < 0) {
            perror("Unable to open GPIO69 value file\n");
            exit (1);
    }

    // fill-in lead A structure
    poll_gpio_fd[A].fd = gpio_val[A];
    poll_gpio_fd[A].events = POLLPRI;
    poll_gpio_fd[A].revents = 0;

    // fill-in lead B structure
    poll_gpio_fd[B].fd = gpio_val[B];
    poll_gpio_fd[B].events = POLLPRI;
    poll_gpio_fd[B].revents = 0;

      // read encoder output values with poll
      while (1) {
            ready = poll(poll_gpio_fd, 2, -1);
            printf("ready: %d\n", ready);
            if (poll_gpio_fd[A].revents != 0) {
                        printf("\t Lead A\n");
                        lead[A] = get_value_lead(gpio_val[A]);
            }
            if (poll_gpio_fd[B].revents != 0) {
                        printf("\t Lead B\n");
                        lead[B] = get_value_lead(gpio_val[B]);
            }
            printf("\t\t A: %d  B: %d\n", lead[A], lead[B]);
      }
      // unexport encoder pins
      gpio_unexport (pin_66);
      gpio_unexport (pin_69);

      return 0;
}
```

```
// ****************************************************************************
//    Function: get_value_lead -- Gets value of encoder ouput A or B
//       fd: value file descriptor
//       return: value of encoder output, 1 or 0
// ****************************************************************************
int get_value_lead(int value_fd) {
      int value;
      int size_buf;
      char buffer[MAX_BUF];

      lseek(value_fd, 0, 0);
      if ((size_buf = read(value_fd, &buffer, sizeof(char))) < 0 ) {
            perror("Unable to read buffer\n");
            value = -1;
            return value;
      }
      buffer[size_buf] = '\0';
      value = atoi(buffer);
      return value;
}

// ****************************************************************************
//    Function gpio_export
//       gpio_num: GPIO pin number
// ****************************************************************************
int gpio_export(char *gpio_num) {

      int exp_fd, write_size;

      // open GPIO export file
      exp_fd = open(sys_gpio_dir"/export", O_WRONLY);
      if (exp_fd == -1) {
            perror("Unable to open export\n");
      }
      // enable GPIO pin
      write_size = write(exp_fd, gpio_num, strlen(gpio_num));
      if (write_size == -1) {
            perror("Unable to write export\n");
      }
      close (exp_fd);
      return 0;
}

// ****************************************************************************
//    Function gpio_set_direction
//       gpio_num:   GPIO pin number
//       dir:            GPIO direction "in"/"out"
// ****************************************************************************
int gpio_set_direction (char *gpio_num, char *dir) {

      int dir_fd, write_size;
      char buffer[64];
      snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/direction", gpio_num);

      // open GPIO direction file
      dir_fd = open(buffer, O_WRONLY);
      if (dir_fd == -1) {
            perror("Unable to open direction file\n");
```

```
        }
        // configure GPIO direction
        write_size = write(dir_fd, dir, strlen(dir));
        if (write_size == -1) {
                perror("Unable to configure the direction\n");
        }
        close (dir_fd);
        return 0;
}

//  ****************************************************************************
//     Function: gpio_set_edge --  Sets GPIO edge
//        gpio_num: GPIO pin number
//        edge: GPIO edge detection "rising"/"falling"/"both"
//  ****************************************************************************
int gpio_set_edge (char *gpio_num, char *edge) {

        int edge_fd , write_size;
        char buffer[MAX_BUF];

        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/edge", gpio_num);

        // open edge file
        edge_fd = open(buffer, O_WRONLY);
        if (edge_fd == -1) {
                perror("Unable to open edge file\n");
        }
        // configure edge event
        write_size = write(edge_fd, edge, strlen(edge));
        if (write_size == -1) {
                perror("Unable to configure the GPIO\n");
        }
        close(edge_fd);
        return 0;
}

//  ****************************************************************************
//     Function: gpio_open_fd -- Opens GPIO value file
//        gpio_num: GPIO pin number
//        return: file descriptor
//  ****************************************************************************
int gpio_open_fd (char *gpio_num) {

        int value_fd;
        char buffer[MAX_BUF];

        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/value", gpio_num);

        // open value file
        value_fd = open(buffer, O_RDONLY);
        if(value_fd == -1) {
                perror("Unable to open value file\n");
        }
        return value_fd;
}

//  ****************************************************************************
//     Function: gpio_unexport
```

```
//         gpio_num:  GPIO pin number
//  ****************************************************************************
int gpio_unexport (char *gpio_num) {

        int unexp_fd , write_size;

        // open GPIO unexport file
        unexp_fd = open(sys_gpio_dir"/unexport", O_WRONLY);
        if (unexp_fd == -1) {
                perror("Unable to open unexport\n");
        }
        // disable GPIO pin
        write_size = write(unexp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write unexport\n");
        }
        close(unexp_fd);
        return 0;
}
```

5.4.2 Proposed exercises

Switch the LED Matrix following the encoder rotation. Start with LD4 switched-on and increase or reduce the number of LEDs switched on accordingly to the rotary encoder movement.

6 Accelerometer

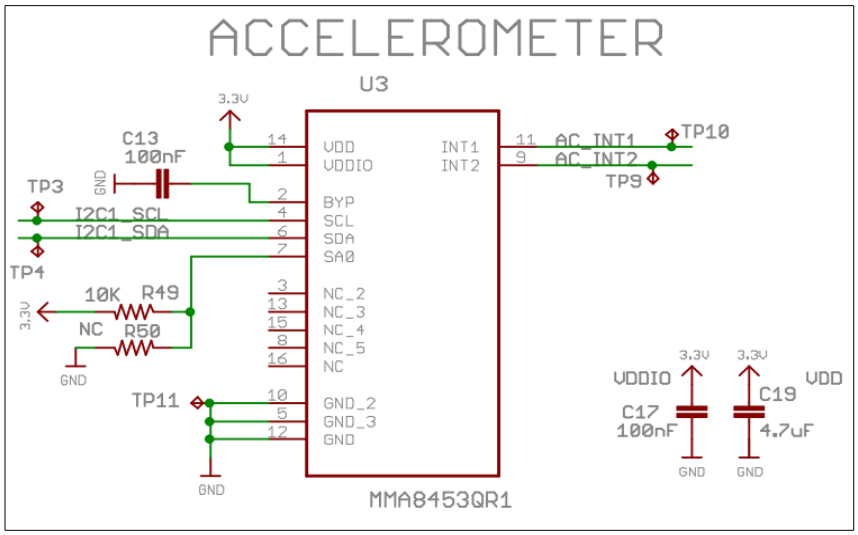
An accelerometer measures the “proper acceleration”, that is, the acceleration an object experience in free fall. Accelerometers can be used to detect orientation, shake, fall, tilt, motion, shock, vibration…

6.1 TT01 Cape Accelerometer

The TT01 cape includes a 3-axis digital accelerometer, reference MMA8453QR1 from Freescale, with 10 bit resolution.

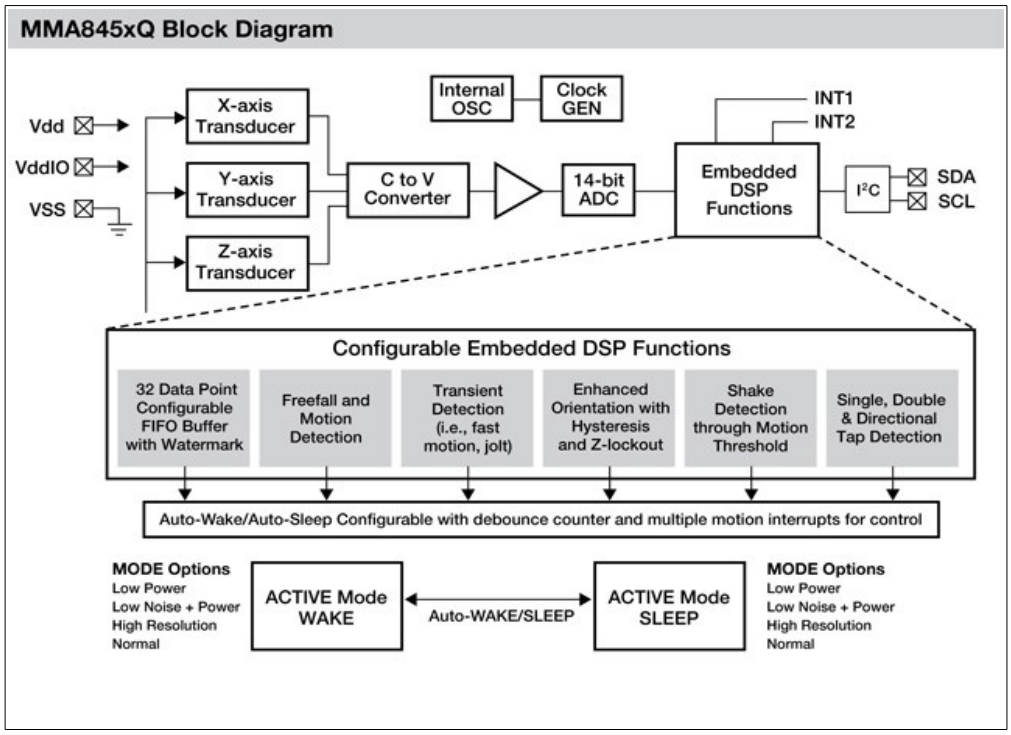
The figure below shows the schematic and signals of the accelerometer.

The I2C1 bus of the Beaglebone is used to communicate with the accelerometer.



This bus has two programmable interrupt signals, AC_INT1 and AC_INT2.

The figure below shows a block diagram of the accelerometer:



6.2 Accelerometer monitoring

We are going to work with the I2C tools provided by Linux. The I2C1 bus can be read with the following shell commands:

```
beaglebone$ cd /lib/firmware
```

```
beaglebone$ i2cdetect -l
   i2c-0   i2c   OMAP I2C adapter   I2C adapter
   i2c-1   i2c   OMAP I2C adapter   I2C adapter
   i2c-2   i2c   OMAP I2C adapter   I2C adapter
```

The `i2c-2` bus corresponds to the Beaglebone I2C1, which is the one connected to the accelerometer and the NFC EEPROM.

We can detect the devices connected to I2C1 with the `i2cdetect` command, specifying what bus do we want to check, in our case `i2c-2`:

```
beaglebone$ i2cdetect -y -r 2
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
   00:          -- -- -- -- -- -- -- -- -- -- -- -- --
   10: -- -- -- -- -- -- -- -- -- -- -- -- -- 1d -- --
   20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   50: -- -- -- 53 -- -- -- 57 -- -- -- -- -- -- -- --
   60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   70: -- -- -- -- -- -- -- --
```

In the data sheet of the accelerometer MMA8453QR1, it is found that the I2C address is 0x1D for SA0='1'.

Once the address is known, we can read the Register Address Map of the accelerometer using the following command:

```
beaglebone$ i2cdump -y 2 0x1d
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   0123456789abcdef
  00: 00 7c 7c 01 fc ed 1c 00 00 00 00 00 3a 00 00   .||????.....:..
  10: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00   .?.D?...........
  20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  80: 00 7c 7c 01 fc ed 1c 00 00 00 00 00 3a 00 00   .||????.....:..
  90: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00   .?.D?...........
  a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

The Register Address Map allows to examine the contents of the accelerometer memory. You can read more about the Register Address Map in the component datasheet (MMA8453Q Rev 5.1, section 6). The Register Address Map contains a set of 8-bit registers, with addresses ranging from 0x00 to 0x7F.

The first registers (addresses 0x01 to 0x06) contain the measured acceleration data in the X, Y and Z axes. Since data size is 10 bits, then each acceleration value spreads in two consecutive registers:

Name	Type	Register Address	Default	Hex Value	Comment
STATUS(1)(2)	R	0x00	00000000	0x00	Real time status
OUT_X_MSB(1)(2)	R	0x01	Output	—	[7:0] are 8 MSBs of 10-bit sample.
OUT_X_LSB(1)(2)	R	0x02	Output	—	[7:6] are 2 LSBs of 10-bit sample
OUT_Y_MSB(1)(2)	R	0x03	Output	—	[7:0] are 8 MSBs of 10-bit sample
OUT_Y_LSB(1)(2)	R	0x04	Output	—	[7:6] are 2 LSBs of 10-bit sample
OUT_Z_MSB(1)(2)	R	0x05	Output	—	[7:0] are 8 MSBs of 10-bit sample
OUT_Z_LSB(1)(2)	R	0x06	Output	—	[7:6] are 2 LSBs of 10-bit sample

By default the accelerometer is in Standby mode. In order to read acceleration values, we must change to Active mode to Active mode. In the datasheet it is found that that the LSB of the control register (address 0x2A) allows to change the mode to Active:

0x2A: CTRL_REG1 System Control 1 Register

0x2A CTRL_REG1 Register (Read/Write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ASLP_RATE1	ASLP_RATE0	DR2	DR1	DR0	LNOISE	F_READ	ACTIVE

In order to change that bit, let us use the command `i2cset`, which has the following format:

```
i2cset <i2cbus> <chip_address> <register_address> <value>
```

So, to change the LSB of the control register (0x2A), execute:

```
beaglebone$  i2cset 2 0x1d 0x2a 0x01
```

Now, if we execute again the dump command, the accelerometer is enabled and therefore we can see some changes in the X-Y-Z acceleration values, highlighted in bold below:

```
beaglebone$  i2cdump -y 2 0x1d
  No size specified (using byte-data access)
```

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   0123456789abcdef
00: ff 01 40 01 00 3f 00 00 00 00 01 00 3a 00 00   .?@?.?.....?.:..
10: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00   .?.D?...........
20: 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00   ..........?.....
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

6.3 Laboratory work

6.3.1 Control with C

In the previous section we have seen a set of Linux tools available to work with I2C buses. However we can work in C with peripherals connected to an I2C bus through a virtual file system, as done in several previous examples.

The following C program illustrates how to work with the accelerometer.

```
//*****************************************************************************
// ACCELEROMETER.c - Displays X,Y,Z axis acceleration data
// Press Push button to exit
//*****************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>
#include <linux/i2c-dev.h>

// function prototypes
int i2c_open();
void i2c_init_slave(int i2c_fd);
void enable_ACCEL(int i2c_fd);
int read_ACCEL(int i2c_fd);
void i2c_close(int i2c_fd);

int gpio_export(char *gpio_num);
int gpio_set_direction(char *gpio_num, char *dir);
int gpio_read_value(char *gpio_num, char *value);
int gpio_unexport(char *gpio_num);

// program constants
#define ACCEL_address  0x1D
#define ACCEL_buffer     0x80

#define MAX_BUF 80
```

```
#define ACC_X_MSB       0x01
#define ACC_X_LSB       0x02
#define ACC_Y_MSB       0x03
#define ACC_Y_LSB       0x04
#define ACC_Z_MSB       0x05
#define ACC_Z_LSB       0x06
#define REG_CTRL_REG1 0x2A

#define I2C1_DIR "/dev/i2c-2"

#define input "in"
#define pin_20 "20" // Push Button - GPIO0_20
#define sys_gpio_dir "/sys/class/gpio/"

int main(void) {
      int i2c1_fd;     // I2C1 file descriptor
      char pb_state[] = "0";   // push button state

      // open I2C1 port
      i2c1_fd = i2c_open();

      // init I2C1 accelerometer
      i2c_init_slave(i2c1_fd);

      // enable accelerometer
      enable_ACCEL(i2c1_fd);

      // export GPIO20 push button and configure it as input
      gpio_export(pin_20);
      gpio_set_direction(pin_20, input);

      // loop to read the X,Y,Z accelerometer variables every 2 seconds
      while (1) {
            // read & display accelerometer values;
            read_ACCEL(i2c1_fd);

            // function to check push button value
            gpio_read_value(pin_20, pb_state);
            // if push button pressed then finish program
            if( strcmp(pb_state,"1") == 0) {
                  printf ("Push button pressed\nExit\n");
                  break;
            }
            sleep(2);
      }
      // unexport GPIO20 (push button)
      gpio_unexport(pin_20);

      // close i2c file and return
      i2c_close(i2c1_fd);
      return 0;
}

//  ***************************************************************************
//    Function: i2c_open -- Opens I2C file
//       i2c_fd: i2c file descriptor
//  ***************************************************************************
int i2c_open() {
```

```
	int i2c_fd;

	if((i2c_fd = open(I2C1_DIR, O_RDWR)) < 0) {
		perror("Open I2C bus failed");
		exit (1);
		}
	return i2c_fd;
}

//  ***************************************************************************
//    Function: i2c_init_slave -- Inits accelerometer as I2C slave
//       i2c_fd: i2c file descriptor
//  ***************************************************************************
void i2c_init_slave(int i2c_fd) {

	if (ioctl(i2c_fd, I2C_SLAVE, ACCEL_address) < 0) {
		perror("Failure to take bus address and communicate with slave");
		exit (1);
	}
}

//  ***************************************************************************
//    Function: enable_ ACCEL -- Enables the accelerometer
//       i2c_fd: i2c file descriptor
//  ***************************************************************************
void enable_ACCEL(int i2c_fd) {

	unsigned char I2C_WR_buffer[MAX_BUF];

	// Accelerometer control register ACTIVE mode value 0x01
	int activeMask = 0x01;

	I2C_WR_buffer[0] = REG_CTRL_REG1;
	I2C_WR_buffer[1] = activeMask;

	if (write(i2c_fd, I2C_WR_buffer, 2) != 2) {
		perror ("Enable Active mode Failure");
		exit (1);
	}
}

//  ***************************************************************************
//    Function: read_ ACCEL -- Read accelerometer data. The
//    register contains the x-axis, y-axis, z-axis 10 bit sample data
//    expressed as 2's complement numbers. This function creates
//    the 10 bit numbers and prints their values as integers.
//       i2c_fd: i2c file descriptor
//  ***************************************************************************
int read_ACCEL(int i2c_fd) {

	int acc_X, acc_Y, acc_Z;
	char RD_buf[7];

	if (read(i2c_fd, RD_buf, 7) != 7){
		perror ("Read error");
	}
```

```
	acc_X=(RD_buf[ACC_X_MSB] << 2)|((RD_buf[ACC_X_LSB] >> 6)& 3);
	acc_Y=(RD_buf[ACC_Y_MSB] << 2)|((RD_buf[ACC_Y_LSB] >> 6)& 3);
	acc_Z=(RD_buf[ACC_Z_MSB] << 2)|((RD_buf[ACC_Z_LSB] >> 6)& 3);

	if (acc_X > 511) acc_X = acc_X - 1024;
	if (acc_Y > 511) acc_Y = acc_Y - 1024;
	if (acc_Z > 511) acc_Z = acc_Z - 1024;

	printf("ACC X: %d\nACC Y: %d\nACC Z: %d\n\n", acc_X, acc_Y, acc_Z);
	return 0;
}

// ***************************************************************************
//    Function: i2c_close -- Closes I2C file
//       i2c_fd: i2c file descriptor
// ***************************************************************************
void i2c_close(int i2c_fd) {

	close(i2c_fd);
}

// ***************************************************************************
//    Function gpio_export
//       gpio_num: GPIO pin number
// ***************************************************************************
int gpio_export(char *gpio_num) {

	int exp_fd, write_size;

	// open GPIO export file
	exp_fd = open(sys_gpio_dir"/export", O_WRONLY);
	if (exp_fd == -1) {
		perror("Unable to open export\n");
	}
	// enable GPIO pin
	write_size = write(exp_fd, gpio_num, strlen(gpio_num));
	if (write_size == -1) {
		perror("Unable to write export\n");
	}
	close (exp_fd);
	return 0;
}

// ***************************************************************************
//    Function gpio_set_direction
//       gpio_num:   GPIO pin number
//       dir:        GPIO direction "in"/"out"
// ***************************************************************************
int gpio_set_direction (char *gpio_num, char *dir) {

	int dir_fd, write_size;
	char buffer[64];
	snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/direction", gpio_num);

	// open GPIO direction file
	dir_fd = open(buffer, O_WRONLY);
	if (dir_fd == -1) {
		perror("Unable to open direction file\n");
```

```
        }
        // configure GPIO direction
        write_size = write(dir_fd, dir, strlen(dir));
        if (write_size == -1) {
                perror("Unable to configure the direction\n");
        }
        close (dir_fd);
        return 0;
}

//  ****************************************************************************
//    Function: gpio_read_value
//       gpio_num:   GPIO pin number
//        value:         gpio value read
//  ****************************************************************************
int gpio_read_value (char *gpio_num, char *value) {

        int value_fd, read_size;
        char buffer[64];
        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/value", gpio_num);

        // open GPIO value file
        value_fd = open(buffer, O_RDONLY);
        if (value_fd == -1) {
                perror("Unable to open value file\n");
        }
        // read GPIO value file
        read_size = read(value_fd, value,1);
        if (read_size == -1) {
                perror("Unable to read input value\n");
        }
        close(value_fd);
        return 0;
}

//  ****************************************************************************
//    Function: gpio_unexport
//         gpio_num:  GPIO pin number
//  ****************************************************************************
int gpio_unexport (char *gpio_num) {

        int unexp_fd , write_size;

        // open GPIO unexport file
        unexp_fd = open(sys_gpio_dir"/unexport", O_WRONLY);
        if (unexp_fd == -1) {
                perror("Unable to open unexport\n");
        }
        // disable GPIO pin
        write_size = write(unexp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write unexport\n");
        }
        close(unexp_fd);
        return 0;
}
```

6.3.2 Proposed exercises

- Configure the Interrupt registers to monitor the interrupts outputs. Only show the X, Y, Z values 2 seconds after a motion detection.
- Display in the LED Matrix the changes in the X, Y, Z values.

```
#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <string.h>

// function prototypes
int get_value_lead(int value_fd);
int gpio_export(char *gpio_num);
int gpio_set_direction(char *gpio_num, char *dir );
int gpio_unexport(char *gpio_num);
int gpio_set_edge(char *gpio_num, char *edge);
int gpio_open_fd(char *gpio_num);

// program constants
#define A 0
#define B 1

#define pin_66 "66" // ENC_A
#define pin_69 "69" // ENC_B

#define input "in"
#define edge_both "both"

#define sys_gpio_dir "/sys/class/gpio"
#define MAX_BUF 64

int main(void)
{
   int gpio_val[2];
   struct pollfd poll_gpio_fd[2];
   int lead[2], ready;

   // export GPIO66 (ENCA) and GPIO69 (ENCB)
   gpio_export(pin_66);
   gpio_export(pin_69);

   // set input direction to GPIO66 and GPIO69
   gpio_set_direction (pin_66, input );
   gpio_set_direction (pin_69, input );

   // set edge of GPIO66 and GPIO69 to both
   gpio_set_edge (pin_66, edge_both);
   gpio_set_edge (pin_69, edge_both);

   // open GPIO66 value file
   gpio_val[A] = gpio_open_fd(pin_66);
   if (gpio_val[A] < 0) {
            perror("Unable to open GPIO66 value file\n");
            exit (1);
```

```
    }

    // open GPIO69 value file
    gpio_val[B] = gpio_open_fd(pin_69);
    if (gpio_val[B] < 0) {
            perror("Unable to open GPIO69 value file\n");
            exit (1);
    }

    // fill-in lead A structure
    poll_gpio_fd[A].fd = gpio_val[A];
    poll_gpio_fd[A].events = POLLPRI;
    poll_gpio_fd[A].revents = 0;

    // fill-in lead B structure
    poll_gpio_fd[B].fd = gpio_val[B];
    poll_gpio_fd[B].events = POLLPRI;
    poll_gpio_fd[B].revents = 0;

      // read encoder output values with poll
      while (1) {
            ready = poll(poll_gpio_fd, 2, -1);
            printf("ready: %d\n", ready);
            if (poll_gpio_fd[A].revents != 0) {
                  printf("\t Lead A\n");
                  lead[A] = get_value_lead(gpio_val[A]);
            }
            if (poll_gpio_fd[B].revents != 0) {
                  printf("\t Lead B\n");
                  lead[B] = get_value_lead(gpio_val[B]);
            }
            printf("\t\t A: %d  B: %d\n", lead[A], lead[B]);
      }
      // unexport encoder pins
      gpio_unexport (pin_66);
      gpio_unexport (pin_69);

      return 0;
}

//  ***************************************************************************
//    Function: get_value_lead -- Gets value of encoder ouput A or B
//       fd: value file descriptor
//       return: value of encoder output, 1 or 0
//  ***************************************************************************
int get_value_lead(int value_fd) {
      int value;
      int size_buf;
      char buffer[MAX_BUF];

      lseek(value_fd, 0, 0);
      if ((size_buf = read(value_fd, &buffer, sizeof(char))) < 0 ) {
            perror("Unable to read buffer\n");
            value = -1;
            return value;
      }
      buffer[size_buf] = '\0';
      value = atoi(buffer);
```

```
	return value;
}

// ****************************************************************************
//    Function gpio_export
//       gpio_num: GPIO pin number
// ****************************************************************************
int gpio_export(char *gpio_num) {

	int exp_fd, write_size;

	// open GPIO export file
	exp_fd = open(sys_gpio_dir"/export", O_WRONLY);
	if (exp_fd == -1) {
		perror("Unable to open export\n");
	}
	// enable GPIO pin
	write_size = write(exp_fd, gpio_num, strlen(gpio_num));
	if (write_size == -1) {
		perror("Unable to write export\n");
	}
	close (exp_fd);
	return 0;
}

// ****************************************************************************
//    Function gpio_set_direction
//       gpio_num:   GPIO pin number
//       dir:        GPIO direction "in"/"out"
// ****************************************************************************
int gpio_set_direction (char *gpio_num, char *dir) {

	int dir_fd, write_size;
	char buffer[64];
	snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/direction", gpio_num);

	// open GPIO direction file
	dir_fd = open(buffer, O_WRONLY);
	if (dir_fd == -1) {
		perror("Unable to open direction file\n");
	}
	// configure GPIO direction
	write_size = write(dir_fd, dir, strlen(dir));
	if (write_size == -1) {
		perror("Unable to configure the direction\n");
	}
	close (dir_fd);
	return 0;
}

// ****************************************************************************
//    Function: gpio_set_edge --  Sets GPIO edge
//       gpio_num: GPIO pin number
//       edge: GPIO edge detection "rising"/"falling"/"both"
// ****************************************************************************
int gpio_set_edge (char *gpio_num, char *edge) {

	int edge_fd , write_size;
```

```
        char buffer[MAX_BUF];

        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/edge", gpio_num);

        // open edge file
        edge_fd = open(buffer, O_WRONLY);
        if (edge_fd == -1) {
                perror("Unable to open edge file\n");
        }
        // configure edge event
        write_size = write(edge_fd, edge, strlen(edge));
        if (write_size == -1) {
                perror("Unable to configure the GPIO\n");
        }
        close(edge_fd);
        return 0;
}

//  ****************************************************************************
//    Function: gpio_open_fd -- Opens GPIO value file
//       gpio_num: GPIO pin number
//       return: file descriptor
//  ****************************************************************************
int gpio_open_fd (char *gpio_num) {

        int value_fd;
        char buffer[MAX_BUF];

        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/value", gpio_num);

        // open value file
        value_fd = open(buffer, O_RDONLY);
        if(value_fd == -1) {
                perror("Unable to open value file\n");
        }
        return value_fd;
}
//  ****************************************************************************
//    Function: gpio_unexport
//        gpio_num:  GPIO pin number
//  ****************************************************************************
int gpio_unexport (char *gpio_num) {

        int unexp_fd , write_size;

        // open GPIO unexport file
        unexp_fd = open(sys_gpio_dir"/unexport", O_WRONLY);
        if (unexp_fd == -1) {
                perror("Unable to open unexport\n");
        }
        // disable GPIO pin
        write_size = write(unexp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write unexport\n");
        }
        close(unexp_fd);
        return 0;
}
```

6.3.3 Proposed exercises

Switch the LED Matrix following the encoder rotation. Start with LD4 switched-on and increase or reduce the number of LEDs switched on accordingly to the rotary encoder movement.

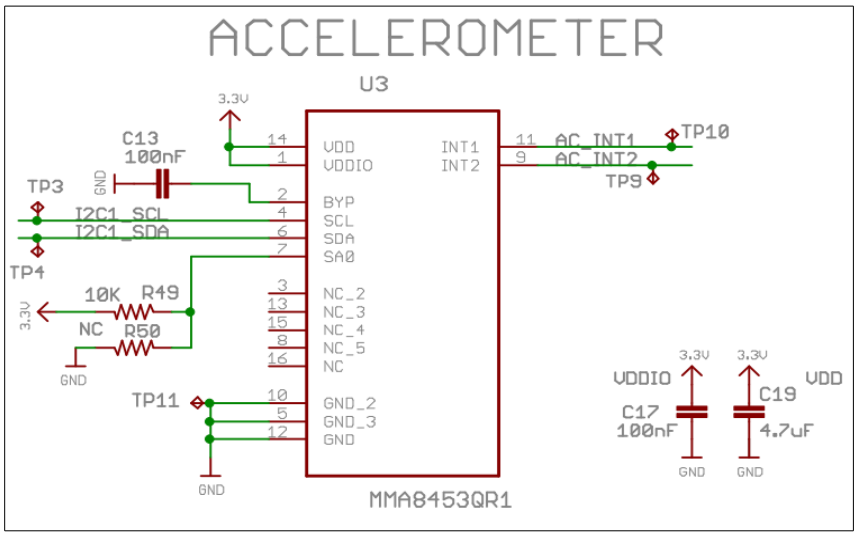
7 Accelerometer

An accelerometer measures the “proper acceleration”, that is, the acceleration an object experience in free fall. Accelerometers can be used to detect orientation, shake, fall, tilt, motion, shock, vibration...

7.1 TT01 Cape Accelerometer

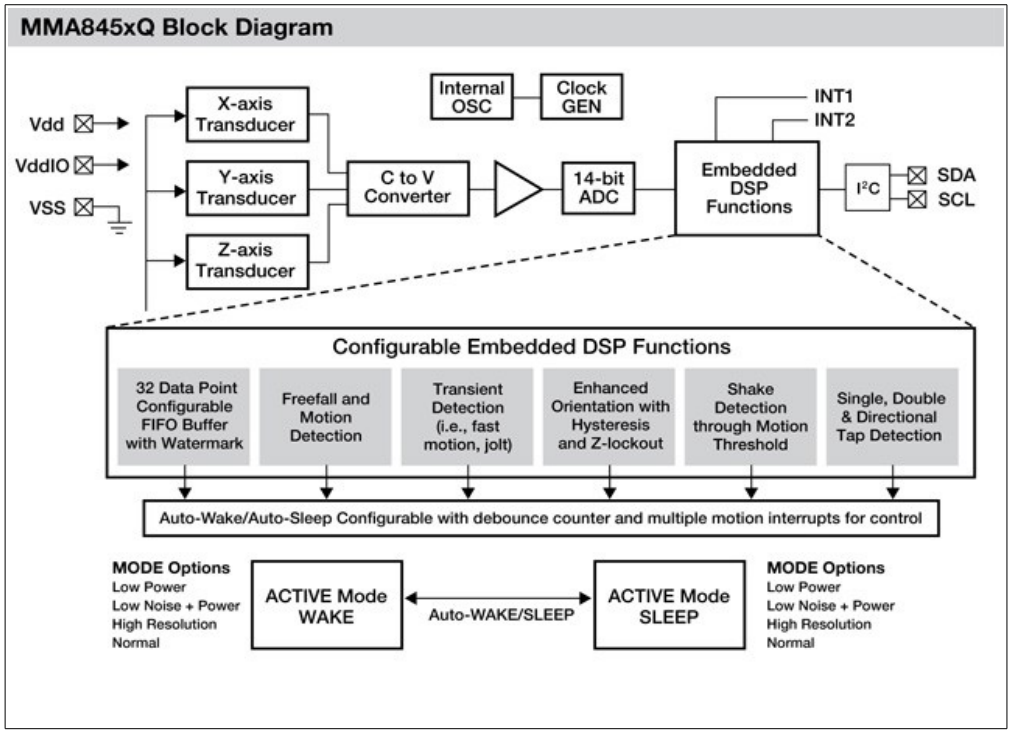
The TT01 cape includes a 3-axis digital accelerometer, reference MMA8453QR1 from Freescale, with 10 bit resolution.

The figure below shows the schematic and signals of the accelerometer.



The I2C1 bus of the Beaglebone is used to communicate with the accelerometer. This bus has two programmable interrupt signals, AC_INT1 and AC_INT2.

The figure below shows a block diagram of the accelerometer:



7.2 Accelerometer monitoring

We are going to work with the I2C tools provided by Linux. The I2C1 bus can be read with the following shell commands:

```
beaglebone$ cd /lib/firmware
beaglebone$ i2cdetect -l
   i2c-0   i2c   OMAP I2C adapter   I2C adapter
   i2c-1   i2c   OMAP I2C adapter   I2C adapter
   i2c-2   i2c   OMAP I2C adapter   I2C adapter
```

The `i2c-2` bus corresponds to the Beaglebone I2C1, which is the one connected to the accelerometer and the NFC EEPROM.

We can detect the devices connected to I2C1 with the `i2cdetect` command, specifying what bus do we want to check, in our case `i2c-2`:

```
beaglebone$ i2cdetect -y -r 2
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
   00:          -- -- -- -- -- -- -- -- -- -- -- -- --
   10: -- -- -- -- -- -- -- -- -- -- -- -- -- 1d -- --
   20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   50: -- -- -- 53 -- -- -- 57 -- -- -- -- -- -- -- --
   60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   70: -- -- -- -- -- -- -- --
```

In the data sheet of the accelerometer MMA8453QR1, it is found that the I2C address is 0x1D for SA0='1'.

Once the address is known, we can read the Register Address Map of the accelerometer using the following command:

```
beaglebone$ i2cdump -y 2 0x1d
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   0123456789abcdef
00: 00 7c 7c 01 fc ed 1c 00 00 00 00 00 00 3a 00 00   .||????......:..
10: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00 00   .?.D?...........
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
80: 00 7c 7c 01 fc ed 1c 00 00 00 00 00 00 3a 00 00   .||????......:..
90: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00 00   .?.D?...........
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

The Register Address Map allows to examine the contents of the accelerometer memory. You can read more about the Register Address Map in the component datasheet (MMA8453Q Rev 5.1, section 6). The Register Address Map contains a set of 8-bit registers, with addresses ranging from 0x00 to 0x7F.

The first registers (addresses 0x01 to 0x06) contain the measured acceleration data in the X, Y and Z axes. Since data size is 10 bits, then each acceleration value spreads in two consecutive registers:

Name	Type	Register Address	Default	Hex Value	Comment
STATUS[1][2]	R	0x00	00000000	0x00	Real time status
OUT_X_MSB[1][2]	R	0x01	Output	—	[7:0] are 8 MSBs of 10-bit sample.
OUT_X_LSB[1][2]	R	0x02	Output	—	[7:6] are 2 LSBs of 10-bit sample
OUT_Y_MSB[1][2]	R	0x03	Output	—	[7:0] are 8 MSBs of 10-bit sample
OUT_Y_LSB[1][2]	R	0x04	Output	—	[7:6] are 2 LSBs of 10-bit sample
OUT_Z_MSB[1][2]	R	0x05	Output	—	[7:0] are 8 MSBs of 10-bit sample
OUT_Z_LSB[1][2]	R	0x06	Output	—	[7:6] are 2 LSBs of 10-bit sample

By default the accelerometer is in Standby mode. In order to read acceleration values, we must change to Active mode to Active mode. In the datasheet it is found that that the LSB of the control register (address 0x2A) allows to change the mode to Active:

0x2A: CTRL_REG1 System Control 1 Register

0x2A CTRL_REG1 Register (Read/Write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ASLP_RATE1	ASLP_RATE0	DR2	DR1	DR0	LNOISE	F_READ	ACTIVE

In order to change that bit, let us use the command `i2cset`, which has the following format:

```
i2cset <i2cbus> <chip_address> <register_address> <value>
```

So, to change the LSB of the control register (0x2A), execute:

```
beaglebone$  i2cset 2 0x1d 0x2a 0x01
```

Now, if we execute again the dump command, the accelerometer is enabled and therefore we can see some changes in the X-Y-Z acceleration values, highlighted in bold below:

```
beaglebone$  i2cdump -y 2 0x1d
  No size specified (using byte-data access)
       0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   0123456789abcdef
  00: ff 01 40 01 00 3f 00 00 00 00 01 00 3a 00 00   .?@?.?.....?.:..
  10: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00   .?.D?...........
  20: 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00   ..........?.....
  30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
  70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

7.3 Laboratory work

7.3.1 Control with C

In the previous section we have seen a set of Linux tools available to work with I2C buses. However we can work in C with peripherals connected to an I2C bus through a virtual file system, as done in several previous examples.

The following C program illustrates how to work with the accelerometer.

```
//*****************************************************************************
// ACCELEROMETER.c - Displays X,Y,Z axis acceleration data
// Press Push button to exit
//*****************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>
#include <linux/i2c-dev.h>

// function prototypes
int i2c_open();
void i2c_init_slave(int i2c_fd);
void enable_ACCEL(int i2c_fd);
int read_ACCEL(int i2c_fd);
void i2c_close(int i2c_fd);

int gpio_export(char *gpio_num);
int gpio_set_direction(char *gpio_num, char *dir);
int gpio_read_value(char *gpio_num, char *value);
int gpio_unexport(char *gpio_num);

// program constants
#define ACCEL_address   0x1D
#define ACCEL_buffer    0x80

#define MAX_BUF 80

#define ACC_X_MSB       0x01
#define ACC_X_LSB       0x02
#define ACC_Y_MSB       0x03
#define ACC_Y_LSB       0x04
#define ACC_Z_MSB       0x05
#define ACC_Z_LSB       0x06
#define REG_CTRL_REG1 0x2A

#define I2C1_DIR "/dev/i2c-2"

#define input "in"
#define pin_20 "20" // Push Button - GPIO0_20
#define sys_gpio_dir "/sys/class/gpio/"

int main(void) {
      int i2c1_fd;    // I2C1 file descriptor
      char pb_state[] = "0";   // push button state

      // open I2C1 port
      i2c1_fd = i2c_open();

      // init I2C1 accelerometer
      i2c_init_slave(i2c1_fd);

      // enable accelerometer
      enable_ACCEL(i2c1_fd);
```

```
	// export GPIO20 push button and configure it as input
	gpio_export(pin_20);
	gpio_set_direction(pin_20, input);

	// loop to read the X,Y,Z accelerometer variables every 2 seconds
	while (1) {
		// read & display accelerometer values;
		read_ACCEL(i2c1_fd);

		// function to check push button value
		gpio_read_value(pin_20, pb_state);
		// if push button pressed then finish program
		if( strcmp(pb_state,"1") == 0) {
			printf ("Push button pressed\nExit\n");
			break;
		}
		sleep(2);
	}
	// unexport GPIO20 (push button)
	gpio_unexport(pin_20);

	// close i2c file and return
	i2c_close(i2c1_fd);
	return 0;
}

//  ***************************************************************************
//    Function: i2c_open -- Opens I2C file
//      i2c_fd: i2c file descriptor
//  ***************************************************************************
int i2c_open() {

	int i2c_fd;

	if((i2c_fd = open(I2C1_DIR, O_RDWR)) < 0) {
		perror("Open I2C bus failed");
		exit (1);
		}
	return i2c_fd;
}

//  ***************************************************************************
//    Function: i2c_init_slave -- Inits accelerometer as I2C slave
//      i2c_fd: i2c file descriptor
//  ***************************************************************************
void i2c_init_slave(int i2c_fd) {

	if (ioctl(i2c_fd, I2C_SLAVE, ACCEL_address) < 0) {
		perror("Failure to take bus address and communicate with slave");
		exit (1);
	}
}

//  ***************************************************************************
//    Function: enable_ ACCEL -- Enables the accelerometer
//      i2c_fd: i2c file descriptor
//  ***************************************************************************
void enable_ACCEL(int i2c_fd) {
```

```
	unsigned char I2C_WR_buffer[MAX_BUF];

	// Accelerometer control register ACTIVE mode value 0x01
	int activeMask = 0x01;

	I2C_WR_buffer[0] = REG_CTRL_REG1;
	I2C_WR_buffer[1] = activeMask;

	if (write(i2c_fd, I2C_WR_buffer, 2) != 2) {
		perror ("Enable Active mode Failure");
		exit (1);
	}
}

//  ***************************************************************************
//    Function: read_ ACCEL -- Read accelerometer data. The
//    register contains the x-axis, y-axis, z-axis 10 bit sample data
//    expressed as 2's complement numbers. This function creates
//    the 10 bit numbers and prints their values as integers.
//      i2c_fd: i2c file descriptor
//  ***************************************************************************
int read_ACCEL(int i2c_fd) {

	int acc_X, acc_Y, acc_Z;
	char RD_buf[7];

	if (read(i2c_fd, RD_buf, 7) != 7){
		perror ("Read error");
	}

	acc_X=(RD_buf[ACC_X_MSB] << 2)|((RD_buf[ACC_X_LSB] >> 6)& 3);
	acc_Y=(RD_buf[ACC_Y_MSB] << 2)|((RD_buf[ACC_Y_LSB] >> 6)& 3);
	acc_Z=(RD_buf[ACC_Z_MSB] << 2)|((RD_buf[ACC_Z_LSB] >> 6)& 3);

	if (acc_X > 511) acc_X = acc_X - 1024;
	if (acc_Y > 511) acc_Y = acc_Y - 1024;
	if (acc_Z > 511) acc_Z = acc_Z - 1024;

	printf("ACC X: %d\nACC Y: %d\nACC Z: %d\n\n", acc_X, acc_Y, acc_Z);
	return 0;
}

//  ***************************************************************************
//    Function: i2c_close -- Closes I2C file
//      i2c_fd: i2c file descriptor
//  ***************************************************************************
void i2c_close(int i2c_fd) {

	close(i2c_fd);
}

//  ***************************************************************************
//    Function gpio_export
//      gpio_num: GPIO pin number
//  ***************************************************************************
int gpio_export(char *gpio_num) {
```

```
        int exp_fd, write_size;

        // open GPIO export file
        exp_fd = open(sys_gpio_dir"/export", O_WRONLY);
        if (exp_fd == -1) {
                perror("Unable to open export\n");
        }
        // enable GPIO pin
        write_size = write(exp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write export\n");
        }
        close (exp_fd);
        return 0;
}

//  ****************************************************************************
//    Function gpio_set_direction
//       gpio_num:   GPIO pin number
//        dir:            GPIO direction "in"/"out"
//  ****************************************************************************
int gpio_set_direction (char *gpio_num, char *dir) {

        int dir_fd, write_size;
        char buffer[64];
        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/direction", gpio_num);

        // open GPIO direction file
        dir_fd = open(buffer, O_WRONLY);
        if (dir_fd == -1) {
                perror("Unable to open direction file\n");
        }
        // configure GPIO direction
        write_size = write(dir_fd, dir, strlen(dir));
        if (write_size == -1) {
                perror("Unable to configure the direction\n");
        }
        close (dir_fd);
        return 0;
}

//  ****************************************************************************
//    Function: gpio_read_value
//       gpio_num:   GPIO pin number
//        value:          gpio value read
//  ****************************************************************************
int gpio_read_value (char *gpio_num, char *value) {

        int value_fd, read_size;
        char buffer[64];
        snprintf(buffer, sizeof(buffer), sys_gpio_dir"/gpio%s/value", gpio_num);

        // open GPIO value file
        value_fd = open(buffer, O_RDONLY);
        if (value_fd == -1) {
                perror("Unable to open value file\n");
        }
        // read GPIO value file
```

```
        read_size = read(value_fd, value,1);
        if (read_size == -1) {
                perror("Unable to read input value\n");
        }
        close(value_fd);
        return 0;
}

//  ****************************************************************
//    Function: gpio_unexport
//        gpio_num:  GPIO pin number
//  ****************************************************************
int gpio_unexport (char *gpio_num) {

        int unexp_fd , write_size;

        // open GPIO unexport file
        unexp_fd = open(sys_gpio_dir"/unexport", O_WRONLY);
        if (unexp_fd == -1) {
                perror("Unable to open unexport\n");
        }
        // disable GPIO pin
        write_size = write(unexp_fd, gpio_num, strlen(gpio_num));
        if (write_size == -1) {
                perror("Unable to write unexport\n");
        }
        close(unexp_fd);
        return 0;
}
```

7.3.2 Proposed exercises

- Configure the Interrupt registers to monitor the interrupts outputs. Only show the X, Y, Z values 2 seconds after a motion detection.
- Display in the LED Matrix the changes in the X, Y, Z values.

8 Appendix. TT01 pin-out.

Cape TT01 P8 connector pin out.

BEAGLEBLONE	TT01 CAPE			TT01 CAPE	BEAGLEBONE
GND	GND	1	2	GND	GND
GPIO1_6	GPMC_AD6	3	4	GPMC_AD7	GPIO1_7
GPIO1_2	GPMC_AD2	5	6	GPMC_AD3	GPIO1_3
TIMER4	GPMC_ALE/ENC_A	7	8	GPMC_REN	TIMER7
TIMER5	GPMC_CLE/ENC_B	9	10	GPMC_WEN	TIMER6
GPIO1_13	GPMC_AD13/ENC_P	11	12	GPMC_AD12	GPIO1_12
EHRPWM2B	GPMC_AD9	13	14	GPMC_AD10	GPIO0_26
GPIO1_15	GPMC_AD15	15	16	GPMC_AD14	GPIO1_14
GPIO0_27	GPMC_AD11	17	18	GPMC_CLK	GPIO2_1
EHRPWM2A	GPMC_AD8	19	20	GPMC_CSN2	GPIO1_31
GPIO1_30	GPMC_CSN1	21	22	GPMC_AD5	GPIO1_5
GPIO1_4	GPMC_AD4	23	24	GPMC_AD1	GPIO1_1
GPIO1_0	GPMC_AD0	25	26	GPMC_CSN0	GPIO1_29
GPIO2_22	NC	27	28	NC	GPIO2_24
GPIO2_23	NC	29	30	NC	GPIO2_25
UART5_CTSN	NC	31	32	NC	UART5_RTSN
UART4_RTSN	NC	33	34	NC	UART3_RTSN
UART4_CTSN	NC	35	36	NC	UART3_CTSN
UART5_TXD	NC	37	38	NC	UART5_RXD
GPIO2_12	NC	39	40	NC	GPIO2_13
GPIO2_10	NC	41	42	NC	GPIO2_11
GPIO2_8	NC	43	44	NC	GPIO2_9
GPIO2_6	NC	45	46	NC	GPIO2_7

Cape TT01 P9 connector pin out.

BEAGLEBLONE	TT01 CAPE			TT01 CAPE	BEAGLEBONE
GND	GND	1	2	GND	GND
VDD_3V3EXP	3.3V	3	4	3.3V	VDD_3V3EXP
VDD_5V	NC	5	6	NC	VDD_5V
SYS_5V	5.0V	7	8	5.0V	SYS_5V
PWR_BUT	NC	9	10	NC	SYS_RESETn
UART4_RXD	GPMC_WT0	11	12	GPMC_BE1N	GPIO1_28
UART4_TXD	NC	13	14	LED3	EHRPWM1A
GPIO1_16	LED1	15	16	LED2	EHRPWM1B
I2C1_SCL	I2C1_SCL	17	18	I2C1_SDA	I2C1_SDA
I2C2_SCL	I2C2_SCL	19	20	I2C2_SDA	I2C2_SDA
UART2_TXD	UART2_TXD	21	22	UART2_RXD	UART2_RXD
GPIO1_17	RFWIP/BUSY	23	24	CAN_TXD	UART_TXD
GPIO3_21	AC_INT2	25	26	CAN_RXD	UART_RXD
GPIO3_19	AC_INT1	27	28	SPI1_CS0	SPI1_CS0
SPI1_D0	SPI1_MISO	29	30	SPI1_MOSI	SPI1_D1
SPI1_SCLK	SPI1_SCLK	31	32	VDD_ADC	VDD_ADC
AIN4	NC	33	34	GNDA_ADC	GNDA_ADC
AIN6	NC	35	36	NC	AIN5
AIN2	NC	37	38	NC	AIN3
AIN0	AN_SW	39	40	NC	AIN1
CLKOUT2	PB1	41	42	SP1_CS1/RS	GPIO1_7
GND	GND	43	44	GND	GND
GND	GND	45	46	GND	GND